

Performance Modeling and Mapping of Sparse Computations

Nadya T. Bliss and Sanjeev Mohindra
MIT Lincoln Laboratory, Lexington, MA
{nt, smohindra}@ll.mit.edu

Una-May O'Reilly
MIT Computer Science and Artificial Intelligence
Laboratory, Cambridge, MA
unamay@csail.mit.edu

Abstract

In the past, knowledge processing (anomaly detection, target identification, social network analysis) of sensor data did not require real-time processing speeds. However, the rapid growth in the size of the data and the shortening time scale of the required data analysis are driving the need for applications that provide real-time signal and knowledge processing at the sensor front end. Many knowledge processing techniques, such as Bayesian networks, social networks, and neural networks, have a graph abstraction. Graph algorithms are difficult to parallelize and thus cannot take advantage of multi-core architectures. Many graph operations can be cast as sparse linear algebra operations. While this increases the ease of programming, parallel sparse algorithms are still inefficient. This paper presents a search-based mapping and routing approach for sparse operations. Since finding well-performing maps and routes for sparse operations is a computationally intensive task, the mapping and routing algorithms have been parallelized to take advantage of the Lincoln Laboratory cluster computing capability, LLGrid. Our parallelization of the approach yielded near linear speed up and the mapping and routing results demonstrate over an order of magnitude performance improvement over traditional mapping techniques.

1. Introduction and Motivation

MIT Lincoln Laboratory has been developing techniques for automatic mapping of signal processing applications. The pMapper^[1,2] automatic mapping approach has been demonstrated to be both feasible and effective. However, modeling, program analysis, and mapping techniques have been limited to dense matrix computations. The dense matrix algorithms of traditional signal processing are highly amenable to performance optimization. However, as sensors collect increasingly large quantities of data, moving post-processing algorithms to the sensor front-end allows the application

to run with greater immediacy and significantly diminish the communication required to a back-end processing station.

Post-processing algorithms frequently have a graphical abstraction. In some cases, such as activity detection, the key data structure that is being analyzed is a graph, or collection of vertices and edges, such as in activity detection. Here, typical algorithms performed on the graph include breadth-first search, depth-first search, betweenness centrality calculation, spanning tree calculation, graph isomorphism, etc. In other cases, the problem does not naturally translate into a graph structure, but graphical models are used, such as Bayesian networks, neural networks, partially observable Markov decision processes, factor graphs, and decision trees. Tasks like inference and planning are often performed on graphical models.

The efficiency of algorithms that operate on graphs is often only a small fraction (<0.01) of the peak throughput of a conventional architecture, and the efficiency deteriorates as the graph size increases. This is due to poor spatial and temporal locality of the data. Specifically, when operating on dense data, usually a contiguous block of data can be brought into cache, operated on and written out. On the other hand, graph algorithms have irregular data access patterns and often, computation will occur on a just one element of data, followed by a costly read across the memory hierarchy. In addition to having poor serial performance, graph algorithms are not well suited for parallelization. The problem of irregular access is compounded by its combination with data distribution across remote memories. Thus, the implementation efficiency is likely to diminish even further. As multi-core processors become increasingly prevalent, a solution to this challenge is required.

Our approach to resolving this challenge is based upon a key observation: there exists a duality between graphs and matrices.^[3] Figure 1 illustrates a sample graph G and a corresponding adjacency matrix A . An adjacency matrix A has N_v rows and N_v columns, where N_v is the number of vertices in the graph. A non-zero entry in row

i column j indicates that there is an edge connecting vertex i and j .

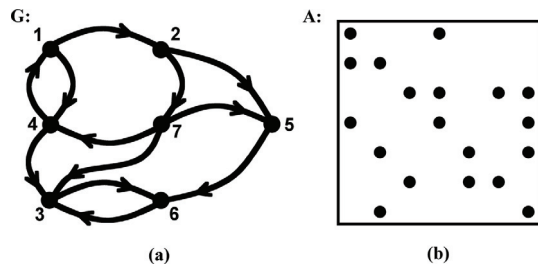


Figure 1. Graph/sparse matrix duality. (a) is a simple directed graph G with 7 vertices. (b) is the corresponding adjacency matrix, A . Non-zero entries in A represent edges in graph G .

A common operation performed on graphs, regardless of whether the problem is formulated as a graph or graphical model, is graph traversal. This operation can be cast as a sparse matrix-matrix multiplication. For many algorithms, the multiplication has to be performed multiple times, and thus the efficiency of an algorithm can be tied to the performance of the sparse matrix-matrix multiplication kernel. For this reason, we have chosen to focus on improving the efficiency of this kernel. While the results presented in this paper focus on matrix-matrix multiplication, our approach is general and could be extended to other sparse array computations.

While sparse matrix formulations expose intrinsic algorithm structure and thus are better suited for parallelization than traditional graph algorithms, the efficiency of sparse matrix computations is still orders of magnitude worse than that of dense. This point is illustrated in Figure 2. The figure compares the performance of a serial dense matrix multiplication with a serial sparse matrix multiplication and shows large discrepancy between the two.

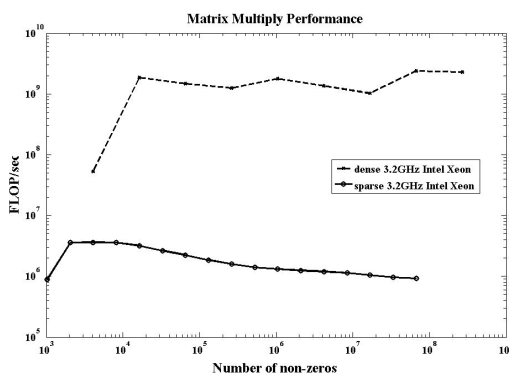


Figure 2. Dense vs. Sparse Matrix Multiplication Performance. Observe that for the same number of non-zeros, the performance of the sparse matrix multiplication is approximately 10^{-3} that of the dense.

The efficiency is expected to deteriorate even further for the parallel implementation, as parallelization introduces the complexity of operating with another layer of memory hierarchy. At this level, not only is the efficient distribution, or mapping, of the sparse matrix multiply kernel necessary, but routing must also be optimized.

The rest of the paper is organized as follows: Section 2 describes our general approach to optimizing sparse matrix multiplication, introduces the map construct, and highlights challenges that make the sparse matrix optimization problem distinct from dense. Section 3 describes the mapping approach—a nested genetic algorithm (GA). Section 4 presents the parallelization of the GA and parallel speedup results on the LLGrid cluster. Section 5 presents performance results for the mappings found using the GA as compared with tradition mapping techniques. Section 6 presents a brief overview of related work. Finally, Section 7 concludes with the discussion and current research directions.

2. Optimizing Sparse Computations

Our general approach to optimizing the sparse matrix multiplication kernel and other sparse computations is finding a set of maps for an array-based computation as was done in the MIT Lincoln Laboratory pMapper project. While this approach restricts the set of programs or sub-programs that can be parallelized, large numbers of graph algorithms can be written in array-based form. The topic of array-based graph algorithms is outside the scope of this paper, however an example of array-based vertex betweenness centrality can be found in Reference 4.

2.1. Maps

A map for an array describes how and where the array is distributed on a parallel hardware architecture. Maps and map-like constructs have a history in High-Performance Fortran^[5], MIT Lincoln Laboratory Space-Time Adaptive Processing Library (STAPL)^[6], ||VSIP++^[7], and Parallel Vector Library (PVL)^[7]. Additionally, a map is a key construct in pMatlab^[8] and Parallel Vector Tile Optimizing Library (PVTOL)^[9]. Figure 3 is an example of a simple two-processor column block map applied to an 8×6 matrix.

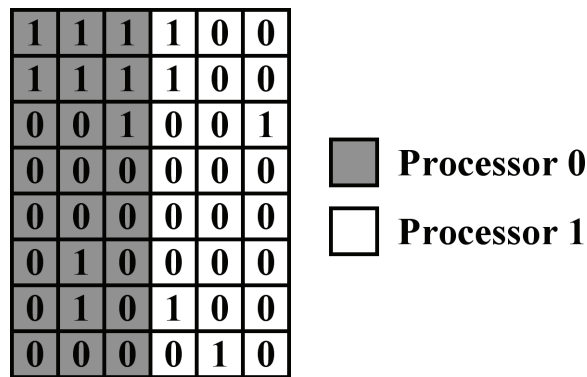


Figure 3. 1D Block map applied to a matrix. Different colors represent different processors—the first 3 columns are mapped onto Processor 0, while the other 3 columns are mapped onto Processor 1.

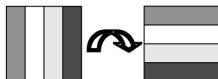



A basic map has three components: grid description, distribution description, and processor list. A map used in Figure 3 can be described as having:

1x2 grid
block distribution
{0,1} processor list

The grid description together with the processor list, specify where blocks of data are distributed on processing elements. Distribution description specifies how the data is distributed. For many applications block, cyclic, and block-cyclic distributions are sufficiently expressive to achieve good parallel performance. Table 1 illustrates common maps used for a number of dense computation kernels.

Block-cyclic distributions are limiting for sparse computations due to the finer granularity of the sparse mapping problem. It is desirable to be able to support arbitrary mappings, without significant increase in index computation cost.

Table 1. Common mappings for dense computations

| KERNEL | MAP |
|-----------------|---|
| FFT | 1D Block  |
| Matrix Multiply | 2D Block  |
| Convolution | Block Overlap  |
| LU | Block Cyclic  |

pMatlab, PVL, and PVTOL maps all use a common underlying index representation: Processor Indexed Tagged FAmiLy of Line Segments (PITFALLS)^[10] The advantage of using PITFALLS is that efficient redistribution algorithms exist, thus yielding fast computation of indices local to each processor and, if the data has to be re-organized, calculation of messages that have to be sent between processors.

To extend the regular block-cyclic map construct and PITFALLS indexing to arbitrary distributions, we allowed the repetition of processors in the processor list. Figure 4 illustrates this with an example. In the worst cast scenario, the common block size is equal to a single element and a processor for each array element must be stored in the processor list. However, if the minimum block size is controlled, index calculations can be performed efficiently.

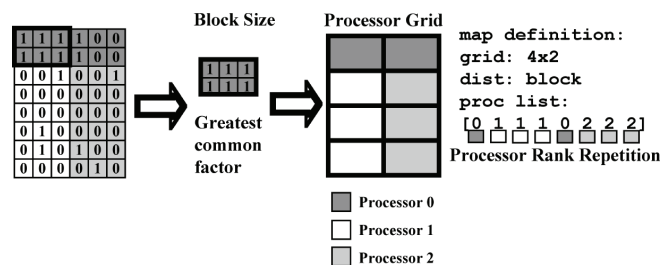


Figure 4. Irregular maps. Support for arbitrary data distributions can be implemented by allowing processor repetition in the processor list.

2.2. Sparse Mapping Challenges

In the previous section, we touched on mapping challenges associated with sparse computations. The first key challenge is the *granularity of the computation*. Specifically, when mapping dense data, it is often sufficient to distribute large blocks of data between processors according to a regular distribution. On the other hand, when mapping sparse computations, much smaller blocks of data must be considered and the regularity of distribution is likely to be detrimental to performance.

The second key challenge is the *granularity of communication*. Consider a dense parallel matrix multiplication. When communication occurs, a large block of data is sent between processors. On the other hand, in a sparse matrix multiplication, a single element often must be communicated between remote processors. Even when the data is not being communicated between two distinct processors but within the memory hierarchy of a single processor, the communication of individual elements incurs a significant overhead.

The third key challenge is that in order to optimize a sparse matrix multiplication or another sparse operation, the *communication and computation*, which are both fine-

grained, *must be co-optimized*. The cost of communication cannot be minimized until the computation is mapped. That is, the options for communication depend on each specific mapping. Thus, neither computation optimization alone nor communication optimization alone will yield sufficient improvement in performance.

2.3. Approach

Our general approach focuses on addressing the three challenges listed in Section 2.2. Specifically, in order to find the best mapping for a sparse matrix multiplication operation, we focus on co-optimizing the efficiency of computation and communication at fine-grain levels.

In Section 2.1, a map construct was presented. Observe, that the map provides distribution information for matrices, however it does not provide routing information. Once the maps for the arrays are defined, the set of communication operations that must occur can be enumerated. However, in order to evaluate the performance of a given mapping, a route for each communication operation must be chosen. Yet, routes cannot be enumerated until a map is defined. Figure 5 illustrates a simplified example with a distributed addition. In the dependency graph, communication operations are highlighted. For each highlighted operation, a number of routing options exists. The number of possible routes is dependent on the topology of the underlying hardware architecture.

Since the best map depends on chosen routes and routes cannot be enumerated until a map is chosen, the problem is combinatorial and no closed form solution exists. This type of problem is well solved by a stochastic search method. Since evaluation of the quality of a solution requires creation and simulation of fine-grained dependency graphs (Figure 5b) on a machine or hardware model, we chose a stochastic search technique well suited for parallelization—genetic algorithm.^[11] The next section describes the problem formulation in greater detail.

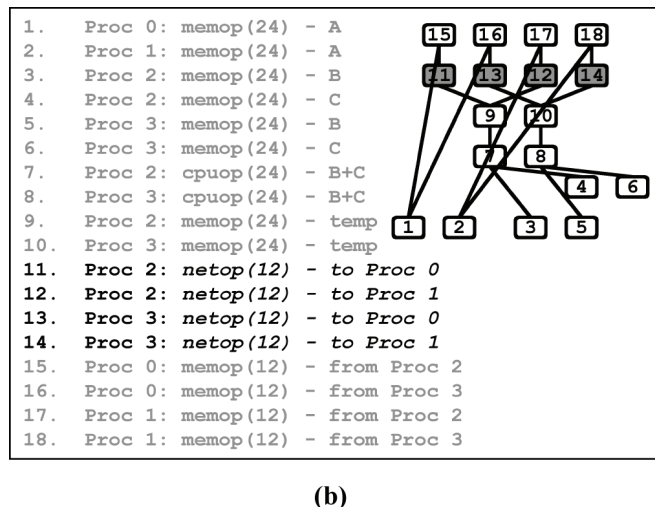
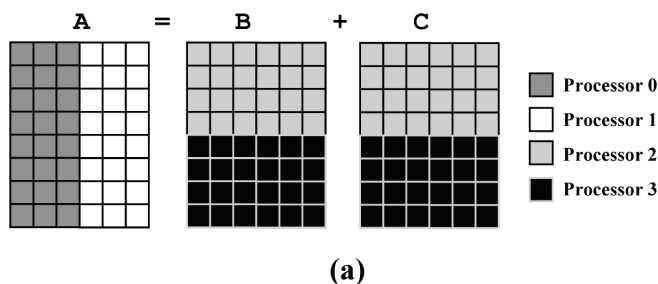


Figure 5. Parallel addition with redistribution. (a) illustrates how arrays A, B, and C are mapped. Note that no communication is required during the addition operation, however communication is required when the result is assigned to A. (b) presents the dependency graph and lists all the memory, communication, and computation operations that must occur. The remote communication operations are highlighted—or each communication operation, a number of possible routes exist.

3. Co-optimization of Mapping and Routing

Figure 6 presents an overview of the nested genetic algorithm (GA) for mapping and routing.

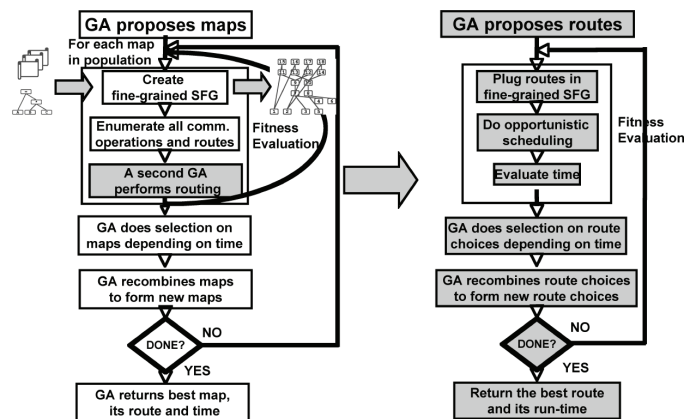


Figure 6. Nested genetic algorithm (GA). The outer GA searches over maps, while the inner GA searches over routes for all communication operations given a map set.

The outer genetic algorithm searches over maps, while the inner GA searches over route options for a given set of maps.

One input to the GA is a coarse-grained dependency graph. The outer GA generates a population of maps for the arrays in the coarse-grain dependency graph. For the matrix-multiply operation $C=A*B$, there are two arrays

for which maps must be generated as C is assumed to have the same map as A . This is a heuristic simplification that does not restrict the generality of solutions found.

For each $\langle \text{maps}, \text{dependency graph} \rangle$ pair, a fine-grained dependency graph is generated. While the original dependency graph contained operations such as matrix multiplication and assignment, the fine-grained dependency graph contains only computation, communication and memory operations, as illustrated in Figure 5b.

Once the fine-grain dependency graph is constructed, the communication operations and corresponding route choices can be enumerated. At this point, the inner GA assigns route choices to communication operations and evaluates the performance of the given $\langle \text{maps}, \text{routes} \rangle$ pairing against a hardware model. Fitness evaluation is performed using opportunistic scheduling and operations in the dependency graph are overlapped whenever possible.

3.1. Representation and Search Space Characterization

3.1.1. Outer GA

The outer GA iterates over sets of maps for arrays in the computation. The minimum block size for a matrix is chosen based on the size of the matrix being mapped. An individual in the outer GA is represented as a set of matrices blocked according to minimum block size. Each block additionally contains information on the sparsity of that block. Processors are assigned to each block and mutation and crossover operators manipulate the processor assignments for each block. Figure 7 illustrates a typical individual for a matrix multiplication operation.

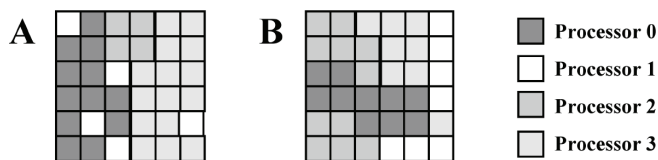


Figure 7. Outer GA individual. Here, the block size yields 36 blocks per matrix. Different shades of gray indicate different processors assigned to blocks. Each block also contains the sparsity information associated with the block. This individual contains two arrays since matrix multiplication requires mapping of two arrays.

3.1.2. Inner GA

The inner GA iterates over routes. The representation for the inner GA is simply the listing of communication operations. The length of the listing is equal to the number of the communication operations for a given set of maps. Each entry in the list represents the

index of a route chosen for a particular communication operation. Figure 8 illustrates an individual for the inner GA.



Figure 8. Inner GA individual. The set of communication operations is represented as a linear array, with each entry in the array containing index of a route chosen for the given communication operation.

3.1.3. Search Space

When performing a stochastic search, it is helpful to characterize the size of the search space. Equation 1 characterizes the search space, S , for the nested genetic algorithm formulation of the mapping and routing problem:

$$S = P^B r^C \quad (1)$$

where

P =number of processors

B =number of blocks

C =number of communication operations

r =average number of route options per communication operations.

4. Parallelization of the Nested Genetic Algorithm

Fitness evaluation of a $\langle \text{maps}, \text{routes} \rangle$ pairing requires building a dependency graph consisting of all communication, memory, and computation operations, performing opportunistic scheduling of operations, and simulating the operations on a machine model. This evaluation is computationally expensive as illustrated by Table 2.

Table 2. Individual fitness evaluation times. Sparsity patterns are described in Section 5. The time shown is average over 30,000 evaluations.

| SPARSITY PATTERN | EVALUATION TIME (MIN) |
|------------------|-----------------------|
| Torroidal | .45 |
| Powerlaw | 1.77 |

Since the sparse mapping and routing framework is written in MATLAB, we used pMatlab to parallelize the nested genetic algorithm and run it on LLGrid: Lincoln Laboratory cluster computing capability.^[12] Figure 9 illustrates the parallelization process.

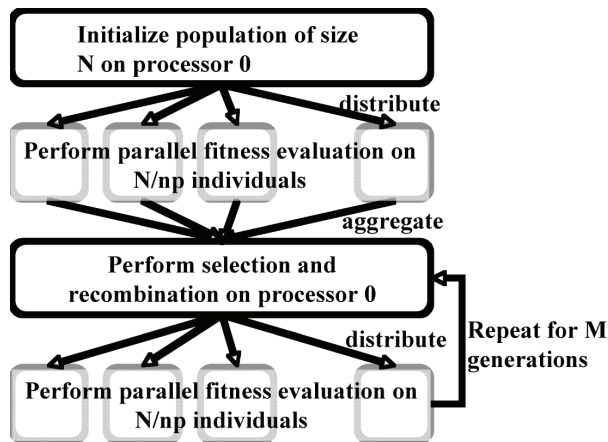


Figure 9. Parallelization process. Fitness evaluation is performed in parallel on np processors, while selection and recombination are performed on the leader processor (processor 0).

As indicated by Figure 9, parallelization required minimal changes to the code (Table 3). A GA is well suited to parallelization, since each fitness evaluation can be performed independently of all other fitness evaluations.

Table 3. Lines of code. Parallelization with pMatlab requires minimal changes to the code.

| | |
|---|-------|
| Nested GA total lines of code | ~1400 |
| Lines of code added for parallelization | ~20 |
| Percentage lines of code added | 1.4% |

4.1. Parallelization Speed-up Results

The parallel code was executed on the TX-2500 cluster of the LLGrid system.^[12] Figure 10 shows the speedup results averaged over a number of different runs of the mapping framework. The selection step occurring each generation of the GA (when parents are chosen for reproduction with likelihood relative to their fitness and fitness of other potential parents) requires global communication. By making sure that each processor had sufficient number of individuals to evaluate, we were able to achieve linear speedup.

LLGrid truly enabled this research by allowing exploration of an optimization space that would be impossible on a single machine.

An important note is that the focus of this iteration of this research was a feasibility study. We wanted to investigate our hypothesis that the fine-grained analysis and co-optimization of communication and computation

via stochastic search over maps and routes was beneficial for parallelization of sparse computations. Currently, we are working on techniques to reduce the computational complexity of the analysis.

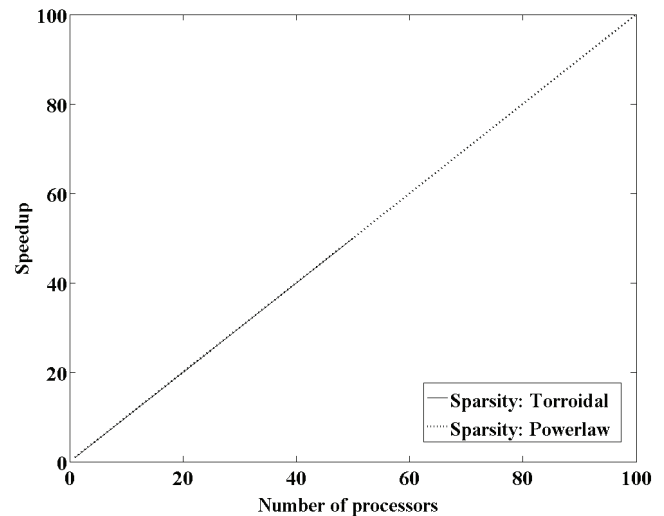


Figure 10. Speedup results. Linear speed up was achieved executing the mapping and routing framework on the LLGrid.

5. Mapping Performance Results

This section discusses the performance results of the maps found using our mapping and routing framework and whether the maps found meet the goals of the research: to find efficient ways of distributing sparse computations onto parallel architectures and gain insight into the type of mappings that perform well.

5.1. Experiment Overview

5.1.1. Machine Model

The results presented here are simulated results on a hardware or machine model. Our framework allows us to alter the machine model freely, thus ultimately we would like to focus on various architecture properties that affect the performance of sparse computations. Table 4 describes the parameters of the model used for the results presented.

Table 4. Machine model parameters

| PARAMETER | VALUE |
|----------------------|----------------|
| Topology | Ring |
| Number of processors | 8 |
| CPU rate | 28 GFLOPS |
| CPU efficiency | 30% |
| Memory rate | 256 GBytes/sec |
| Memory latency | 10^{-8} sec |
| Network rate | 256 GBytes/sec |
| Network latency | 10^{-8} sec |

5.1.2. Matrix Multiplication Algorithm

The mapping and routing framework was applied to an outer product matrix multiplication algorithm.^[13] Figure 11 illustrates the algorithm and the corresponding pseudocode. This algorithm was chosen due to the independent computation of *slices* of matrix C. This property makes the algorithm well suited for parallelization.

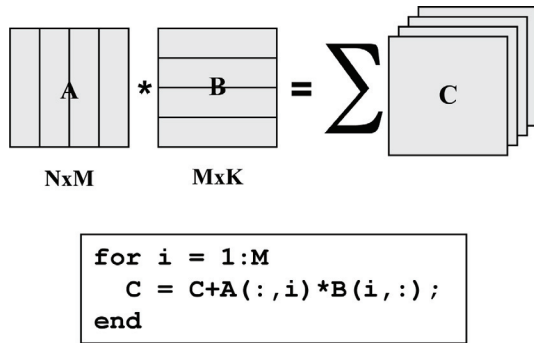


Figure 11. Outer product matrix multiplication

5.1.3. Sparsity Patterns

We wanted our solutions to apply to general sparse matrices and thus we tested the mapping framework on a number of different sparsity patterns. Figure 12 illustrates the sparsity patterns mapped in increasing order of load balancing complexity, from random sparse to scrambled powerlaw.

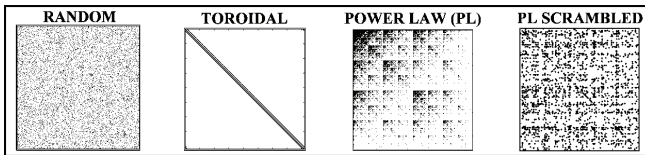


Figure 12. Sparsity patterns

5.1.4. Benchmarks

Figure 13 illustrates a number of standard mappings that the results obtained with the sparse mapping and routing framework were compared against.

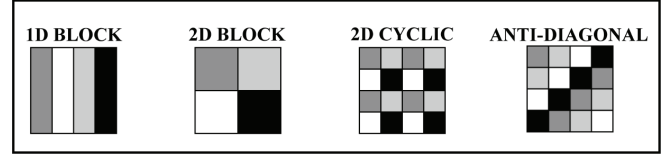


Figure 13. Benchmark maps. We compared our results with the results using standard mappings.

5.2. Results

Figure 14 presents performance results achieved by the mapping framework. Our maps outperform standard maps by more than an order of magnitude. The results are normalized with regards to the performance achieved using a two-dimensional (2D) block-cyclic map, as that is the most commonly used map for sparse computations.

We wanted to make sure that the results achieved were repeatable and statistically significant over a number of runs of the GA. Figure 15 shows statistics for 30 runs of the GA on a powerlaw matrix. Observe that there is good consistency in terms of solution found between multiple runs of the mapping framework.

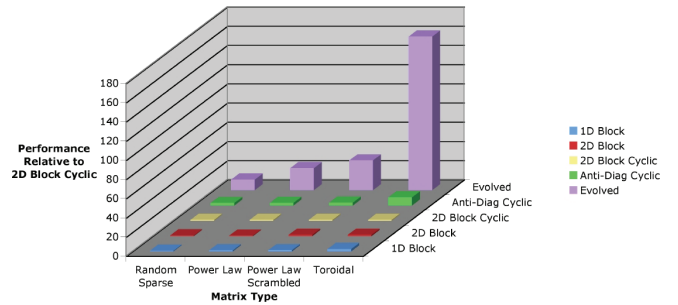


Figure 14. Mapping performance results

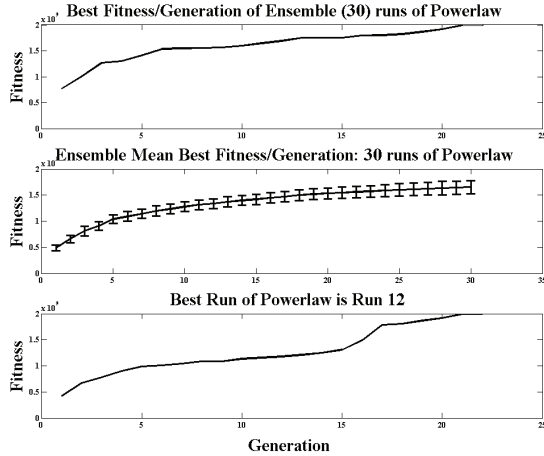


Figure 15. Run statistics. The top plot shows best overall fitness found for each generation. The middle plot shows average fitness for each generation. Finally, the bottom plot shows the behavior of the best of the 30 runs over 30 generations.

Note that while we did consider a large number of possible solutions, we have only explored a small fraction of the search space. For the statistics runs in Figure 15, the outer GA was run for 30 generations with 1,000 individuals. The inner GA used a greedy heuristic to pick the shortest route between two nodes whenever possible. Thus, the total number of solutions considered was:

$$30 \times 1,000 = 30,000$$

The size of the search space per Eq. 1 is

$$S = P^B r^C \sim 8^{128} * 2^{100}$$

where

- 8 is the number of processors in the machine model;
- 128 is the number of blocks used for 256×256 matrices;
- O(100) is the number of communication operations;
- 2 is the number of possible routes for each communication operation given a ring topology.

Thus, the GA performs well in this optimization space, as it is able to find good solutions while exploring rather insignificant fraction of the search space.

6. Related Work

In the domain of sparse matrix-vector multiplication, a lot of work has been done on optimizing these operations using register blocking and cache blocking techniques (e.g., References 14–16). Cache blocking techniques reorder memory accesses to increase temporal locality, and register-blocking techniques compress the data structure to reduce memory traffic. These techniques are well suited for optimizing memory hierarchy access in sequential code. Other work has addressed the issue of

avoiding communication^[17] in sparse matrix computations in both sequential and parallel code.

Our approach is fundamentally different. It is general enough to be applied to any sparse matrix computation, and is not limited to matrix-vector or matrix-matrix multiplication. We decompose the problem of finding optimal spatial and temporal locality, and of optimizing communication from the algorithm itself by use of maps. We parameterize the algorithm using maps and then use our framework to derive the optimal mapping from the structure of the matrix and the needs of the algorithm. We co-optimize the location of data and the routing of communication using nested genetic algorithm to perform the search. This, coupled with our rich machine model, makes our framework well suited for optimizing any sparse matrix computation on any machine or network topology.

7. Conclusions and Discussion

In this paper, we presented an approach for mapping and routing sparse matrix calculations. As this effort started out as a feasibility study, we focused on not constraining the computation costs of our solution. Specifically, we wanted to explore whether fine-grained analysis, irregular distributions, map/route co-optimization, and sparsity-influenced mapping were beneficial to improved performance of sparse computations. LLGrid allowed us to explore optimization spaces that would not be possible otherwise. The results obtained provided over an order of magnitude performance improvement over traditional mapping techniques. Clearly, invoking our current system at runtime would not be beneficial; however, mappings found can be saved for later use. Additionally, we can readily evolve well performing set of maps for various matrix families.

Moving forward, we are focusing on reducing the computation costs of the mapping framework. These include dependency graph re-use and considering other matrix multiply algorithms.

Acknowledgements

The authors would like to extend their thanks to the following for their assistance in supporting this research: William Arcand, William Bergeron, Robert Bond, Tim Currie, Matthew Hubbell, Jeremy Kepner, Andy McCabe, Peter Michaleas, Julie Mullen, Daniel Rabideau, Albert Reuther, and Ken Senne. This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author

and are not necessarily endorsed by the United States Government.

References

1. Travinin, N., H. Hoffmann, R. Bond, H. Chan, J. Kepner, and E. Wong, "pMapper: Automatic Mapping of Parallel Matlab Programs." *HPCMP UGC 2005*.
2. Bliss, N., J. Dahlstrom, D. Jennings, and S. Mohindra, "Automatic Mapping of the HPEC Challenge Benchmarks." *High Performance Embedded Computing (HPEC) Workshop 2006*.
3. Cormen, T.H., C.E. Leiserson, R.L. Rivest, and S. Stein, *Introduction to Algorithms*, Second Edition, McGraw-Hill Book Company, Boston, MA, 2001.
4. Robinson, E., "Array Based Betweenness Centrality." *13th SIAM Conference on Parallel Processing for Scientific Computing*, Atlanta, Georgia, March 2008.
5. Loveman, D.B., "High Performance Fortran." *Parallel and Distributed Technology: Systems and Applications*, IEEE 1(1).
6. DeLuca, C.M., C.W. Heisey, R.A. Bond, and J.M. Daly, "A Portable Object-Based Parallel Library and Layered Framework for Real-Time Radar Signal Processing." *Proc. 1st Conf. International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE '97)*, pp. 241–248.
7. Lebak, J., J. Kepner, H. Hoffmann, and E. Rutledge, "Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing." *Proceedings of the IEEE 93*.
8. Bliss, N.T., and J. Kepner, "pMatlab Parallel MATLAB Library." *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on High-Productivity Programming Languages and Models*, Vol. 21, No. 3, SAGE 2007.
9. Kim, H., et al., "PVTOL: Providing Productivity, Performance and Portability to DoD Signal Processing Applications on Multicore Processors." *HPCMP UGC 2008*.
10. Ramaswamy, S. and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers." *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers '95)*, McClean, VA, February 6–9.
11. Mitchell, M., *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, 1998.
12. Reuther, A., J. Kepner, A. McCabe, J. Mullen, N. Bliss, and H. Kim, "Technical Challenges of Supporting Interactive HPC." *HPCMP UGC 2007*.
13. Golub, G.H. and C.F. Van Loan, *Matrix Computations*, 3rd Edition, John Hopkins University Press, Baltimore, MD, 1996.
14. Sparsity, <http://www.cs.berkeley.edu/~yelick/sparsity/>.
15. Nishtala, R., R. Vuduc, J. Demmel, and K. Yelick, "When Cache Blocking of Sparse Matrix Vector Multiply Works and Why." *Applicable Algebra in Engineering, Communication, and Computing*, Vol 18, Issue 3, 2007.
16. Vuduc, R., "Automatic Performance Tuning of Sparse Matrix Kernels." Ph. D. Thesis, University of California, Berkeley, 2003.
17. Demmel, J., M. Hoemmen, M. Mahiyuddin, and K. Yelick, "Avoiding Communication in Sparse Matrix Computations." *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.